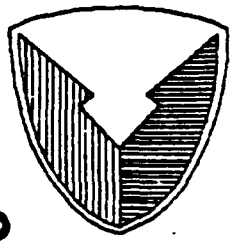
(2)



CECOM

CENTER FOR SOFTWARE ENGINEERING

ADVANCED SOFTWARE TECHNOLOGY

Subject: **Final Report - Ada Tasking Performance Issues**

CIN: **C02 043NW 0001 00**

23 February 1990

ADA TASKING PERFORMANCE ISSUES

FINAL REPORT

PREPARED BY:
LabTek Corporation
8 Lunar Drive
Woodbridge, CT 06525

PREPARED FOR:
Battelle
Research Triangle Park Office
200 Park Drive
P.O. Box 12297
Research Triangle Park, NC 27709

SPONSORING AGENCY:
U.S. Army HQ CECOM
The Center for Software Engineering
Fort Monmouth, NJ 07703-5000

DATE:
9 February 1990

CONTRACT NUMBER: DAAL03-86-D-0001
DELIVERY ORDER 1258
Scientific Services Program

Table of Contents

Table of Contents

Table of Contents

1. Introduction

This document reports on the findings of the task titled "Ada Tasking Performance Issues". The work was sponsored by the U.S. Army HQ CECOM, Center for Software Engineering, Ft. Monmouth, NJ, through the U.S. Army Research Office, Scientific Services Program.

The objectives of this task were to determine, for both uni-processor and distributed real-time applications, the sufficiency of the Ada tasking model in relation to the performance needed and to recommend ways of improving the efficiency of the Ada tasking model to obtain the needed performance.

1.1 Project Scope

The project scope was as follows:

1) Determine the performance issues involved with using the Ada tasking model for uni-processor applications,
2) Determine the performance issues involved with using the Ada tasking model for distributed applications,
3) Analyze the strengths and weaknesses of the Ada tasking model and recommend methods to overcome any lack of performance,
4) Determine how to best utilize the Ada tasking model for both uni-processor and distributed applications, AND
5) Propose experiments to test the conclusions reached in this research.

1.2 Report Organization

The report is organized as follows:

The "Approach" taken to this task is described in section 2.0.

Section 3.0 begins the "Technical Discussion". It begins with background and definition material (section 3.1), which may be read as necessary or skipped entirely.

Section 3.2 describes the "Performance Issues in a Uni-processor Application" and provides a taxonomy of the tasking services for a uni-processor application. It reports on example instruction counts for each tasking service for the studied runtimes.

Section 3.3 describes the "Performance Issues in a Distributed Application" and expands on the taxonomy of section 3.2 to provide a taxonomy of the tasking services for a for a distributed system.

Section 3.4 describes the "Strengths & Weaknesses of the Ada Tasking Model" and describes several optimizations which could be used with an evaluation of each one.

Section 3.5 describes the "Best Utilization of the Ada Tasking Model for Uni-Processor Applications".

Section 3.6 describes the "Best Utilization of the Ada Tasking Model for Distributed Applications".

Section 4.0 provides a "Summary of the Results, Recommendations and Conclusions" reached in this report.

Section 5.0 describes "Further Experiments" which could be undertaken to test the conclusions of this report.

Section 6.0 lists the "References" used in this report.

Appendix A is a description of the "Ada Tasking Model" and can be consulted as necessary.

2. Approach

The approach taken for the steps outlined in the project scope (section 1.1) is detailed below.

A taxonomy of processes required to implement the semantics of the Ada tasking model for a uni-processor application was produced. A description of what is entailed with each component of the taxonomy is provided. A runtime was selected and studied to assess the execution time expended (typical case) for each of the elements of the taxonomy. These execution times were compared against the corresponding execution time for what is considered (by experienced users) to be a high performance runtime.

The taxonomy produced above was then extended to include the processes required to implement distributed rendezvous. Special attention must be taken to isolate communication overhead due to software protocols, hardware delays, and Ada semantics. Issues such as reliable communication, hardware message acknowledgement, and shared network resources are discussed.

The strengths of the Ada tasking model are interspersed throughout the document where relevant. The weaknesses of the Ada tasking model are discussed in detail in section 3.4. A discussion and evaluation of *Optimization Techniques* is presented there.

Finally, a discussion of how to best utilize the Ada tasking model for both uni-processor and distributed applications is presented.

A method for evaluating the conclusions reached in this report is described in section 5.0, along with a test environment and appropriate benchmarks to assess the capabilities and relative merit of the recommended solution.

3. Technical Discussion

3.1 Background Definitions

3.1.1 Concurrency

This section introduces the fundamentals of concurrency, including common methods of supporting it in programming languages.

A task is a program unit that may run simultaneously with other program units. Execution of a program containing two or more tasks can be physically concurrent or logically concurrent. Physically concurrent means that each task runs on its own processor. Each task is said to have its own distinct thread of control. (Programs that do not contain tasks are sequential, and have a single thread of control.) Logically concurrent means that there are fewer processors than tasks, and programs are executed by means of some form of processor sharing, such as time slicing. The threads of control of these tasks are interleaved. In most of this paper, there is no distinction between logical and physical concurrency. When the distinction is required, it is clearly stated. A disjoint task is one that carries out its execution without interaction with any other program units. The only language support mechanisms required of a disjoint task are those to specify its computation, initiate that computation, and terminate execution. Few of the valuable applications of concurrent programming are able to use only disjoint tasks. When tasks must interact with other tasks, the required support mechanisms, both in the language design and in the runtime system, are complex.

3.1.2 Task Interaction

Task interaction can take two different forms, data communication and synchronization. Data communication among tasks can be accomplished by two methods, shared variables and message passing. Message passing can also be used as a form of synchronization. Synchronization interaction between two tasks is usually required for data communication, but is also required in situations where no data is shared. In simple terms, synchronization means that the two synchronizing tasks must be at particular respective spots in their executions at the same point in time. To achieve synchronization, the task that reaches its synchronization point first must wait there for the other task to arrive at its corresponding synchronization point. Generally speaking, synchronization is required if two or more tasks are to work together toward a common computational goal.

3.1.2.1 Data Communication

3.1.2.1.1 Shared Variables

Shared variables provide an indirect method of communication among tasks. These variables reside in a program unit outside the tasks that wish to share the data. All communication is through the outside program unit, which is usually another task. At this point, a classical and yet common problem for tasks, the producer-consumer problem, is introduced. It is solved by use of a shared storage area called a buffer. Two tasks, the producer and the consumer, must access the buffer. The producer places data in the buffer and the consumer removes data from the buffer. The buffer is usually implemented as a data structure consisting of a circular queue, a variable for the number of filled positions in the buffer, and includes indices to the next empty position and the position from which the next element to be removed should be taken. While the simple case of a single producer

and a single consumer is often considered, all of the following discussion applies to the more general problem of multiple producers and multiple consumers all sharing the same data buffer.

To illustrate one form of synchronization required between a producer and a consumer, suppose that both are simultaneously accessing the buffer. Suppose further that the producer is slightly ahead of the consumer, and has already placed its data in the buffer and is in the process of incrementing the variable that records the number of elements currently in the buffer. Assume the name of this variable is FILLED. Suppose that the process, at the machine level, of incrementing FILLED is to fetch the value from its memory location and place it in a register, add one to the register's value, and place the new value back in memory. Suppose that the value of FILLED has been fetched from memory into the register by the producer, and at that point in time the consumer task begins its process of decrementing FILLED. It, of course, first fetches the value from the memory location of FILLED, which at that time is the same value that was fetched by the producer task. Both tasks continue their computation of FILLED, with the consumer's value being the last stored in FILLED. Because both began with the same value, the resulting value is one less than the original value (before either modified it). The correct value is the same as the original, so an error has occurred. This error is due to a lack of *competition synchronization*. The two tasks compete for access to the shared variable, FILLED, and these accesses must be synchronized. Competition synchronization is provided by forcing mutually exclusive access to shared data, usually by some form of locks or guards on the shared data. A second kind of error can occur with the shared buffer. This is a simple matter of buffer overflow or underflow. These are errors due to the lack of *cooperation synchronization*. The producer task must be prevented from inserting a value into a full buffer, and the consumer task must be prevented from removing a value from an empty buffer. Cooperation synchronization is

provided by some form of guard, which prevents access to shared data by tasks that would destroy the integrity of the data. Note that cooperation synchronization is a problem that is not limited to concurrent programs.

3.1.2.1.2 Message Passing

Message passing is a direct, explicit form of data communication. Data is transmitted directly from one task to another, or possibly in both directions. No intermediate program unit is needed. A different kind of synchronization is required for a task to send another task a message. This synchronization is explained in Section 3.1.2.2.3. Message passing is clearly a more natural method than data sharing when the tasks are distributed over a network of processors that do not share memory. On the other hand, message passing requires more runtime overhead than data sharing, although both methods require synchronization among the involved tasks.

3.1.2.2 Methods of Providing Synchronization

There are three distinct methods of providing the kinds of synchronization required for the effective use of concurrency: semaphores, monitors, and message passing. Message passing, therefore, is a process that can provide both data communication and synchronization among tasks.

3.1.2.2.1 Semaphores

Semaphores were developed by Dijkstra (1968) as a method of providing both competition and cooperation synchronization. They are primitive devices consisting of a non-negative integer and a queue capable of storing task identifiers. Semaphores are abstract data types that can be changed only by two predefined operations, WAIT and SIGNAL. (WAIT was

originally called P after a Dutch word, and SIGNAL was similarly called V. Some people use the name SEND instead of SIGNAL.) Assuming semaphore variables are defined as records with the fields COUNT and QUEUE, the operations of WAIT and SIGNAL can be defined as:

```
WAIT (SEM)   : if SEM.COUNT > 0 then
                 SEM.COUNT := SEM.COUNT - 1
               else suspend caller and place it in SEM.QUEUE
               end if

SIGNAL (SEM) : if SEM.QUEUE is empty then
                 SEM.COUNT := SEM.COUNT + 1
               else move a process from SEM.QUEUE to ready
               end if
```

Note that "ready" in the SIGNAL definition means the queue of tasks that are available for execution when a processor becomes available. Semaphores can be used to provide both competition and cooperation synchronization for the shared buffer of the producer-consumer problem. Let the producer use a routine named INSERT to place its data in the buffer and let the consumer use a routine named REMOVE to remove data. The calls to INSERT and REMOVE must be synchronized, as they provide the access to the shared buffer. These calls are called the critical regions of the producer and consumer tasks. Three semaphores can be used to protect the critical regions: ACCESS, to provide mutually exclusive access for competition synchronization; FULLSPOTS, to guard against underflow; EMPTYSPOTS, to guard against overflow. For example, in the producer task access to the buffer could appear as:

```
...
WAIT (EMPTYSPOTS);  -- wait for an empty spot in buffer
WAIT (ACCESS);  -- wait for exclusive access to the buffer
INSERT (DATA);  -- go ahead with the access
SIGNAL (ACCESS);  -- release control of the buffer
SIGNAL (FULLSPOTS);  -- increase the number of full spots
...
```

Note the two distinct uses of semaphores here. EMPTYSPOTS is used to count the empty spots in the buffer and also to delay the producer when there are no empty spots available. This provides cooperation synchronization. ACCESS, on the other hand, is used to provide mutually exclusive access to the buffer. Its counter counts the number of waiting tasks, rather than a parameter of the buffer. Semaphores are an adequate method of supporting concurrency. However, they are low-level devices that can be easily misused in ways that neither the compiler nor the runtime system can detect. For example, if the user simply forgets to include the SIGNAL(ACCESS) statement in the above code, no further access to the buffer will occur. This is deadlock caused by the failure of competition synchronization.

3.1.2.2.2 Monitors

A monitor is a syntactic device to encapsulate the accesses to a shared data structure. A monitor is, in fact, an abstract data type whose objects are to be shared by tasks. A monitor for the shared buffer of the producer-consumer problem would contain the buffer as local data, along with the INSERT and REMOVE procedures for accessing the buffer. The key point of the semantics of a monitor is that the procedures contained in the monitor are, by definition, executed in mutual exclusion. The implementation is required to provide this mutually exclusive execution. Therefore, the user need not be concerned with competition synchronization when using a monitor to control access to shared data. This is far more convenient and clearly more reliable than using semaphores. Use of a monitor for the shared buffer of the producer-consumer still requires user action to provide cooperation synchronization. Rather than semaphores, languages that provide monitors, such as Concurrent Pascal (Brinch Hansen, 1975), also provide operations for cooperation synchronization. In Concurrent Pascal, these are called *delay* and *continue*, which block and release the execution of the tasks which call them. For example, *delay (client_q)* suspends

the calling task and places it in a task queue named client_q, which is used to store waiting client tasks. The critical region of the entry procedure has the form:

```
...
if buffer is full then
  delay (client_q);
end if;
-- put the data item in the buffer
continue (server_q);   -- allow a server task to run
...
```

In this code, server_q is a queue for storing waiting server tasks. As stated above, monitors are more reliable than semaphores, because of the implicit mutual exclusion. That is only half of the need for semaphores in synchronizing access to shared data, however, and a language with monitors still requires the use of some sort of guards to provide cooperation synchronization. In the case of Concurrent Pascal, these guards are very similar to semaphores, and share their reliability problems. Monitors first appeared in the literature in the early 1970s (Brinch Hansen, 1973).

3.1.2.2.3 Message Passing

Message passing as a technique for supporting concurrent programs originated in papers by Brinch Hansen (1978) and Hoare (1978). A message with parameters communicates data, and a message without parameters can be used for simple synchronization. The basic process of sending a message between a client task and a server task is as follows. The client task attempts to send the message. If the server is already waiting for the message, the message transmission takes place. If the server is busy when the message is sent, the client must wait until the server is willing to accept the message. Both the client and the server must be ready before the message transmission takes place. The time period during which the two tasks communicate is called a rendezvous. A rendezvous may involve two-way data transfer, with the server also transmitting information back to the client. Messages are sent to specific

locations in the receiving task, which can be thought of as message sockets. A task can have any number of message sockets. There can be contention when multiple messages have been sent to a task and are waiting to be accepted. The question of how this situation is handled is answered in Appendix A for the Ada version of message passing. Message passing is a more general technique than monitors. In fact, monitors can be constructed with message passing. Semaphores can also be implemented with message passing. However, this generality is not without its cost. There is more overhead associated with message passing systems than with either semaphores or monitors, both in the compiled code and in the runtime support system.

3.1.3 Deadlock

There is one more condition that can occur in concurrent programs that must be mentioned here, deadlock. Deadlock is an event in a concurrent program that prevents further progress in its execution. To illustrate deadlock, suppose that there exists a program with two shared data structures, A and B, and two tasks, X and Y, both of which need A and B. Assume that some kind of locks are used to provide competition synchronization for A and B. Now, if task X requests and locks A and, simultaneously, task Y requests and locks B (before X requests it) then deadlock occurs. Task X waits indefinitely for B and task Y waits indefinitely for A. All progress stops, permanently, in both tasks. Deadlock is obviously a serious threat to the reliability of a program, and therefore concern must be given to its prevention by language designers, language implementers, and users.

3.2 Performance Issues in a Uni-Processor Application

The Ada tasking model (as described in Appendix A) contains a variety of concurrency control features. These include: selective wait, guarded **accept** statements, **else**, **delay** alternatives, **terminate** alternatives, conditional **entry** calls, timed **entry** calls, **abort**, priorities, and task allocators.

The runtime for Ada tasking includes routines which support these language features. They can be broken into a taxonomy as follows:

Task Initiation - Performed whenever the declarative region containing a task object is elaborated, or when an allocator of a task type is evaluated. Task initiation creates a new thread of control and elaborates the task body. It may vary slightly depending on whether the task has been statically declared at the library level, declared within a subprogram, or created because of the execution of an allocator for a task type. Runtime support for task initiation includes:

- Establishing the task type.
- Establishing the Master of the task and maintaining a data structure to associate the Master with its dependents.
- Creating a task control block, and initializing it for use.
- Entering a task into the activation list for a particular declarative region or allocator.
- Activation of the tasks in the activation list. This implies elaborating the task body and placing the task on the ready queue.
- Finally, setting the task status to callable and not terminated.

If an exception occurs during the above steps, the exception is changed to TASKING_ERROR and it is raised in the master.

Task Termination - Ends the execution and reclaims the resources of a task. Task termination is invoked by:

- Normal completion of the task body by reaching the last "end" statement.
- Completion due to an unhandled exception.
- Completion due to servicing an **abort** request.

Abort requires making a task abnormal, and depending upon the implementation, possibly waiting before actually terminating task execution. An abnormal task reaching a synchronization point requires that it be terminated. Since possibly several tasks can appear in an abort statement, an abort list may be generated prior to commencing with the destruction of the tasks.

Runtime support for task termination includes:

- Support for each of the termination methods.
- Waiting for dependent tasks to become completed.
- Reclaiming storage used for dynamic task objects.
- Checking for completion of the master, which allows the master to be exited.
- Setting status information so that 'Callable and 'Terminated are FALSE and TRUE, respectively.

Delay and Time Services - Provide the ability to access and manipulate date and time information as well as suspend execution of a task for a specified duration. Delay statements and references to the predefined package Calendar invoke these services. Runtime support for **delay** and time services include:

- Hardware support (as a minimum) for interrupts at a specified time.
- Ability to service clock interrupts to resume tasks suspended because of **delay**. (This includes timed entries and **delay** alternatives too.) Frequently interrupts occur at a fixed interval which are counted to result in **delay** expirations after the correct period.

Advanced Synchronization and Communication is supported by:

- conditional **entry** calls;
- timed **entry** calls;
- simple selective wait;
- selective wait with **else**;
- selective wait with **delay**; and
- selective wait with **terminate**.

These typically have unique runtime routines to support their particular behavior. In addition to the same services required by the basic rendezvous, they also require complex intertask coordination capabilities. For example, there is a need to maintain a list of open **accept** alternatives so that only **entry** calls to those alternatives are considered. Obviously, the timed **entry** calls and **delay** alternatives depend on the delay services specified in the delay and time services above. Guard evaluation is done by the generated code, however, the final set of open alternatives must be conveyed to the runtime.

Scheduling - Provides the ability to choose which task should execute at the proper time. The only language structure for user control over scheduling is use of the priority pragma, however implementation defined pragmas and compilation system options often permit additional capabilities. Runtime support for scheduling is generally considered to include:

- "Ready" queue support which maintains a list of tasks available to execute, usually prioritized to provide fast selection of the highest priority ready task.

- Preemption support which allows a task execution to be suspended at a (nearly) arbitrary point, and allow a higher priority task to run. The suspended task is later resumed at the point of preemption. This feature imposes several requirements on the design for much of the runtime since it implies reentrancy and the mutual exclusion primitives necessary to guard access to shared data structures.

- Custom scheduling services are often provided to support features such as time-slicing, or biased scheduling (providing a task with a percentage of CPU utilization).

- In a sense, the support for interrupt entries is part of the scheduling services, since interrupt entries are treated as very high priority **entry** calls. Therefore interrupt handling is placed under scheduling within the taxonomy and requires support for preemption as indicated above.

Attribute Evaluation - Supports the determination of task characteristics at execution time. Three task attributes are supported which generally have runtime routines specifically to support them. These are:

- COUNT - provides the number of tasks currently trying to call an **entry**.
- CALLABLE - provides a boolean response indicating if a task can be called, (that is, it has been activated and has not terminated).
- TERMINATED - provides a boolean response indicating if a task has terminated.

3.2.1 Performance Assessment of an Example Runtime System

The following table provides instruction counts for an example runtime system as exercised with a set of example programs. Many of the programs were selected from the Performance Issues Working Group (PIWG) test suite so as to study those cases perceived to be of greatest interest. Note that the first runtime shown was not considered a high performance runtime. For comparison, selected tasking services from one of the highest performance runtimes were also measured. The chart below indicates a wide range of performance for similar functions. However, no two runtimes appear to divide the processing up exactly the same way. Variations can occur in what is done as generated code and what is left to the runtime. Also differences can be found in what is done by partners in a rendezvous and how task creation, initialization, activation, and elaboration are handled. The primary differences between these two examples are that the low performance version was coded largely in Ada, sometimes required a context switch to a supervisor state, and was not optimized. The high performance runtime was written entirely in assembly language and special cases used separate routines rather than calling common subroutines. It was highly optimized and required no additional context switches to a supervisor task state.

Tasking Services	Low Performance Example Instruction Count	High Performance Example Instruction Count
Task creation	3,389	233
Establishing a task master	284	not available
Complete_Activation	757	not available
Task Activation	931	481
Master_Complete	623	not available
Abort Statement (single task)	1,423	not available
Calling an Entry	1,135	143
Open accept alternative	194	not available
Wait for alternative to be selected	489	150
Completion of Rendezvous	820	not available
Exception within accept body	159	not available
Delay Statement	1,407	not available
Elaborate the body of a Task	141	not available
Evaluate 'callable attribute	139	not available
Evaluate 'terminated attribute	133	not available
Evaluate 'count attribute	215	not available
Initialize Entry Family	163	not available
Establish a task type	212	not available
Enter on abort list	221	not available

3.3 Performance Issues in a Distributed Application

Support for distributed Ada adds several additional functions to the taxonomy described above. These include:

Communication Subsystem - Consists of the underlying networking mechanisms that allow the distributed system to synchronize and transfer information. This implies one or more communication channels such as Ethernet, or 1553 bus, as well as the software necessary to provide reliable communications across the channel(s). If acknowledgements are used, this involves additional overhead (see below under Acknowledgement Control). Priority observance and time stamp support are very important characteristics of the subsystem, but are frequently only marginally supported.

Message Transmission - Involves the collection of information necessary in the message, such as entry parameters; locating the destination address by using a task directory; and delivering the packetized message to the communication subsystem. Overhead for each message transmission primarily consists of copying data and processing the interrupt generated when the message has been sent. Time stamping of the transmission time is beneficial for time synchronization, but is ideally done by the communication subsystem.

Message Reception - Refers to the allocation of a buffer to receive the message data and identifying the task to get the message. The semaphore used to indicate arrival of the message must be signaled and the message enqueued. It may be necessary to time stamp received messages so as to reduce the uncertainty during time synchronization, however this is ideally done by the communication subsystem.

Timed Message Transfer - To accurately synchronize independent clocks within a distributed system, message transmission and reception times must be determined precisely.

Typically, the actual transport time across the communication medium is deterministic within some small uncertainty. To prevent this uncertainty from becoming large, the times at which the message is sent and received must be recorded (time stamped) with respect to the clocks in the nodes performing the operations.

Acknowledgement Control - Each regular message that is sent must be acknowledged to insure reliable arrival. If either the message or its acknowledgement is lost, retransmission is invoked to try to recover from the loss. A list of outstanding acknowledgements is maintained and as acknowledgements arrive, they are removed from the list. If one is outstanding beyond an allowable limit, this would be an error condition (see timeout detection below).

Timeout Detection - To detect the loss of a message, a timer must be (effectively) set for each normal message transmission. If no acknowledgement is received before the timer expires, error recovery must be initiated. Error recovery is handled by the retransmission mechanism to retry the message. In addition to normal message acknowledgements, "heartbeat" messages may be sent simply to insure that the other system nodes are "alive". These heartbeats are designed to detect node failures even when no normal traffic exists between nodes and are acknowledged like regular messages. They may be sent continuously at some rate, or only when other traffic is not present.

Retransmission - Error recovery typically involves retransmission of "lost" messages. Retransmissions for a particular message are typically counted, and after a specified number of failures the second stage error recovery routines are invoked to take more aggressive action. This is usually the invocation of a runtime routine to raise an exception or even begin system reconfiguration.

Message Sequencing/Buffer Control - When acknowledgements are lost because of network transient errors, the sending node will timeout and retransmit the message. To prevent the same message from being replicated during this type of failure, each message is sequenced. When a sequence number is received that has already arrived, that message is acknowledged and simply discarded. Additionally, due to errors or network topologies that have multiple store-and-forward points, it may be possible for messages to arrive in a different order than the transmitted order. The sequencing insures that the messages are processed in the order of transmission from source nodes independent of any exchanges that occur. This implies a buffering mechanism that can maintain out of order reception and deliver them in the proper order. Finally, special buffer control may be necessary to prevent overrun on some systems.

Communication Failure Detection and Propagation (firm failures) - Some network architectures allow for the positive determination of hardware failure. In these cases, the distributed runtime should make available this additional information for the benefit of application level recovery routines. Since the failure has been isolated to the communication system, the possibility of the other processors continued operation is more likely than a known processor failure or an unclassified error type. An example of this is the collision detection circuitry used for packet collisions on Ethernet. If a very long sequence of collisions occur, this is an indication of network failure. Other networks provide remote loopback which insures that communications are operational to a remote node, even if the processor is not responding.

Global Time Synchronization and Maintenance - The time at which a periodic synchronization message is sent can be recorded at the point it is sent and at the point that it is received (see ed Message Transfer, above). These are taken with respect to different

clocks and compared. The skew in clock values can be determined within a small error limit and eliminated by adjusting the designated clock(s). Since the clocks will run at different rates, the synchronization will need to be done periodically, depending on the allowable error. Clock synchronization is often necessary to make meaningful use of time sensitive data that is shared among the nodes in the system. Also, the Ada semantics for timed **entry** calls and selective waits with **delay** alternatives can be implemented if effective synchronization can be achieved.

The execution time for distributed Ada support can be divided into three areas:

- Ada semantics,
- protocol overhead,
- hardware overhead (transmission time, interrupt latency, etc.).

All of the standard uni-processor tasking semantics must be supported on the distributed system and for the most part can be optimized in the same way as uni-processor Ada. However, to preserve the Ada semantics for timed rendezvous additional actions are required. The normal, "ready for rendezvous check" can become a series of messages and context switches. Also, support of failure semantics can require additional overhead to support the ability to raise an exception within an **accept** body at any arbitrary point because of a communication failure.

A substantial overhead can be consumed in supporting the protocols necessary for insuring that messages are generated, packetized, sequenced, delivered, received, and acknowledged. Typically, this aspect of the distributed system is the most time consuming, but this is largely dependent on the underlying network system. A dedicated system may have very simplistic protocols (basically Destination/Source information only) and provide hardware for message acknowledgement or failure indications and guaranteed order-of-arrival. In these cases, the overhead can be minimized using efficient buffer control mechanisms which make

packet construction very fast. Other cases may require various operations to be performed for each message transferred, such as:

- checksums to be computed,
- conversion of data to a standard format,
- setting and cancelling of timeouts for message acknowledgement,
- switching to privileged mode,
- copying of data several times,
- complex algorithms to prevent destination buffer overrun,
- dividing messages into small packets,
- and resequencing packets that could be out of order due to transmission failures and resends.

Often the time synchronization must be supported as well by the protocol mechanism by using various time-stamping techniques and periodic resynchronizing based on the accuracy of the independent clocks. By computing drift rates between the clocks individual processors can adjust their clocks and therefore resynchronization is required much less frequently.

The hardware component of the overhead can be substantial if the amount of data to be transmitted is large or latencies in obtaining network access are potentially large. Communication subsystems in typical use vary from 10K bits/second to 100M bits/second. Access latencies can vary widely as well, with most real-time networks currently focusing on token passing schemes to make the latency predictable. Essentially none of the popular networks provide support for maintaining accurate time, access to global memory, or good support for gaining access based on priority. Transport times for networks such as Ethernet are a minimum of approximately 60us for the smallest possible message and increase by 800 ns per byte transferred. However, access to the network may take tens of milliseconds or longer depending on network loading. This unpredictability makes contention networks such as Ethernet unsuitable for serious real-time work unless network loading is well understood and controlled.

The performance concerns for Ada tasking in uni-processors (or even tightly coupled multi-processors) are compounded by distribution. Since the tasking model for Ada uses only synchronous communication, it requires a substantial amount of interaction among tasks. This is exactly the type of activity that is difficult to achieve in a distributed system. The ability to have buffers which receive data and provide elasticity in the synchronization of concurrent processes is especially necessary in distributed systems. Ada can provide this utilizing "buffer tasks" but unless the buffer's thread of control is optimized away, the overhead is usually unacceptable.

Conventional distributed systems use mailbox communication mechanisms where the transfer of the message is acknowledged in hardware, but the destination task need not be involved in the transfer. Generally the mailbox buffer is designed to be large enough to handle worst case requirements, otherwise the communicating tasks can throttle their messages or compensate for lost messages.

Ada "high level" task communication structures complicate the messaging system further. Timed and conditional **entry** calls, selective waits with **delay** or **terminate** alternatives all pose serious problems with the distribution of Ada programs. Conventional network based distributed systems generally provide only simple packet transfer services, often with little or no timing services. These networks are ill-suited to support Ada because they lack capabilities upon which Ada runtimes fundamentally depend. These services include:

- Prioritization of services
- Guaranteed worst case response times
- Time synchronization support (stamping)
- Fast access to global state information (availability to rendezvous).

Although conventional networks will be suitable for distributed Ada in environments that are not time-critical, use for real-time systems will be restricted to very high performance

networks such as the 100 Mbps Fiber Distributed Data Interface (FDDI). Only these networks provide sufficient performance so that the communication overhead and uncertainty can be tolerated. Ideally, an Ada specific network could be developed that would provide efficient services for Ada tasking. Unfortunately, it is unclear if such a network would be commercially viable. Obviously, it would have to make use of the same components utilized in networks such as FDDI, but utilize a different protocol interface. Work done previously [22] on this issue described a star topology network with an active hub. This hub would contain system-wide memory and clocks, as well as atomic primitives for manipulating those objects. This study on improving tasking performance has reinforced the conclusions drawn regarding the need for a custom network architecture. Essentially, Ada tasking is extremely complex when compared to other forms of concurrency controls; it will need hardware cooperation to achieve good performance.

To illuminate the complexity associated with distributed tasking, an example is useful. A good case to study is a case where a timed **entry** call is made to a task which is executing a selective wait with a **delay** alternative. In this case, a message must be sent by the caller to the server to initiate its part of the rendezvous. A message must be sent back to acknowledge receipt of the message. When the server is ready for a rendezvous it sends a message back to the caller indicating its willingness to commit, this message must also be acknowledged. Finally, the caller must agree to the commitment or send back a denial. As usual this must be acknowledged as well. Therefore, it generally takes six (6) messages just to initiate a complex rendezvous. It is possible in some cases to reduce the count by combining acknowledgments with other data, but this results in other problems such as the need to acknowledge acknowledgements, which complicates the protocol. As usual, the end of rendezvous will also result in a message and acknowledgement bringing the total up to eight(8). At 200us per message this requires 1.6 milliseconds of overhead as compared to

the equivalent 1us test and set operation on a system where some memory is shared by both tasks. The problems that arise are largely due to the uncertainties in communication delays, and that interaction with the timeouts in the conditional rendezvous from both ends of the transaction. If the clocks can be sufficiently synchronized, it may be desirable to have the both delays measured by the server. Since the underlying communication system must issue "heartbeat" transfers to detect failures during a rendezvous, there is no additional overhead for using this mechanism as the failure detection mechanism prior to the entry being accepted.

3.4 Weaknesses of the Ada Tasking Model

3.4.1 The Cost of Ada Task Execution

The cost of Ada task execution has been the cause of a great deal of concern about the viability of the language for real-time applications, which are exactly the applications for which Ada was intended.

A real-time performance benchmark study [27] was performed from which the cost of various operations of task execution, relative to procedure calls can be illustrated. The hardware used for benchmarking was a Sun [1] 3/60 CPU running Sun UNIX [2] 4.2 Release 3.5, linked to a single 12.5 Mhz Motorola 68020 single board computer enclosed in a multibus chassis. The compiler was the Verdix Ada Development System [3] targeted to Motorola MÇ68020 [4] targets, release 5.41.

---

[1]Sun is a registered trademark of Sun Microsystems, Inc.

[2]UNIX is a registered trademark of AT&T.

[3]Verdix is a registered trademark of Verdix

[4]MC68020 is a registered trademarks of Motorola.

The subprogram overhead benchmark passed an Integer parameter: mode **in out**.

The task elaboration, activation, termination benchmark had a task type and task object defined in a procedure which was declared in a package. The procedure was called from the main program. Entry into the procedure activated the task object and the procedure exited when the task terminated.

The complex rendezvous benchmark had a main program calling an entry in another task with one Integer parameter: mode **in out**.

The times represented below are in microseconds.

```
Procedure call time.....................   9.7
Task activation and termination time.... 4800.0
Parameterized rendezvous time..........  355.4
Ratio of rendezvous to procedure call...  36.6
```

The ratios of parameterized rendezvous and procedure calls are strikingly high. The obvious question, particularly for a user who is not knowledgeable in compilation problems for tasks, is: How can a rendezvous possibly take between one and two orders of magnitude more time than a procedure call. After all, they should be quite similar: Both require some sort of switch of referencing environment and some register and status saving. Studies indicated that the vast majority of work being performed during a rendezvous is not associated with register save and restore, but rather in implementing the complex semantics of the rendezvous.

3.4.2 Sources of Task Overhead

The rendezvous is probably the most important part of Ada tasking to consider when one is concerned with overhead, because it can happen many times during the execution of a program, as opposed to task creation, which occurs only a relatively small number of times.

The primary overhead events of rendezvous are context switches and message selection. A state switch is a change of control between two program units. It requires saving and restoring machine status and registers, parameter transmission, and the saving and formation of a referencing environment. State switches are required for procedure calls. A context switch is a state switch plus actions for transfer to a different thread of control. Message selection, or more directly, the execution of the **select** statement, can incur a significant cost, particularly when the number of alternatives is large and a significant proportion of them have guards. The presence of **terminate** and **delay** alternatives are also significant contributors to the cost of message selection. For example, before a **terminate** alternative can be selected, the status of the task's master and all its dependent tasks must be checked. To accommodate this process, a "master" list of other tasks, which includes the task's master, along with all of the other tasks dependent on the master task, must be maintained. The status of the tasks on the list determine whether the **terminate** alternative can be selected. Because tasks can be dynamically created and can also terminate during execution, this list is dynamic. Every task creation and every task termination require modification to the list. Complex programs require a complex master list, whose maintenance requires a significant amount of computational overhead. Programs that contain tasks must be run with a stack that is, at least logically, tree-shaped. These are usually called cactus stacks. The cactus stack is required because each task needs an area of stack for any data it allocates from the stack, and also for the activation records of subprograms it calls. Management of the cactus stack may not require a great deal of computation, but it can certainly require significant amounts of memory. The primary additional costs, then, of a task call over a subprogram call is in blocking the caller, scheduling the called task, and in message selection.

3.4.3 Proposals for Increasing the Speed of Rendezvous

3.4.3.1 Operational Assumptions

The definition of Ada suggests that the code in an accept statement in a task is executed under the control of that task. This report refers to this implementation technique as the "normal" technique. The normal method is outlined in the Ada design rationale (Ichbiah, et al., 1979). In the following section several alternative approaches are discussed. In the remainder of this report, the possibility of time slicing which adds context switches to task execution, is ignored. It is assumed that tasks always run as long as they can, which means until they are blocked or preempted by a newly available task with higher priority.

There have been several significant investigations of alternative techniques for implementing Ada tasks, with the common goal of achieving faster rendezvous. In this section the results of these efforts are reviewed and evaluated. Note that there may have been other related studies, but, if so, a literature search failed to produce them. Any study of the kind reflected in the remainder of this report faces certain difficulties. The most significant of these has its source in the competitiveness of the Ada compiler business. While there undoubtedly has been a great deal of effort expended in optimization of the rendezvous, no compiler vendor is likely to publish its best ideas on the subject. The problem with this for the Ada community is that work on optimizations is duplicated over and over among different vendors, rather than researchers advancing the work of others. Another problem with this kind of study is the difficulty of evaluating various optimization approaches. Two of the most effective means of evaluation, prototype construction and simulation, are obviously beyond the scope of this work.

3.4.3.2 A Normal Implementation

First, the number of context switches required by a normal implementation, as described in Section 3.4.2.2 are examined. A very simple execution scenario is assumed, in which there are only two tasks, a client task and a server task. The client task is assumed to repeatedly call the server. Finally, it is assumed that there are no interruptions of the process; that is, no input or output interrupts occur. Such interrupts can greatly complicate the analysis. Consider a single rendezvous between the client and the server task. If the client arrives first at the **entry** call possibly two context switches are required to complete the rendezvous. As soon as the client attempts the call, it is blocked, requiring a context switch. Then, when the server completes the execution of the rendezvous, the client is moved to the ready queue, and is scheduled for execution, possibly requiring another context switch. This is only required if the priority of the client is greater than the server. If the server task arrives at the **accept** statement first, it is blocked, causing a context switch. When the client arrives, it is blocked, causing a second context switch. The server is put in the ready queue to allow it to eventually execute the **accept** statement statements. When the rendezvous is completed, the client must be put in the ready queue. Once again, if its priority is higher than that of the server another context switch is required. This analysis of a single rendezvous indicates that two context switches may be required if the client arrives first and three may be required if the server arrives first.

3.4.3.3 Habermann and Nassi

An early study of techniques for increasing the speed of the rendezvous was done by Habermann and Nassi (1980). This work includes proposals for two different implementation methods.

3.4.3.3.1 Letting the Caller Execute the Rendezvous

The first proposal of Habermann and Nassi is an implementation method for simple rendezvous. In this case, the support for the technique could easily be written for the runtime, because the situations in which it can be applied are simple to detect. The idea is simply to attempt to reduce the number of context switches required for a rendezvous by letting the calling task execute the **accept** statement. Habermann and Nassi believed that by allowing the client to execute the rendezvous, rather than the server, fewer context switches could be required. Based on this belief, they then developed a method of having code for rendezvous that operates that way.

3.4.3.3.2 Conversion of a Server to a Monitor

The second proposal in the Habermann and Nassi paper is a process whereby a server task is implemented as a monitor. This transformation removes all context switches, effectively eliminating some of the blocking and scheduling. In fact, the only remaining blocking is caused by the remaining need for mutually exclusive execution in server tasks. The process of transforming the server task to a monitor is outlined, but no method of automating the conversion is offered. To convert the server task to a monitor, all code to implement the **select** statement operation is logically moved into each **accept** statement. The process is designed to work only if there is no code outside **accept** statements, except for the possible code that precedes the outermost **select** statement. Logically, then, when a client task executes an **entry** call on the server task, the client task not only executes the rendezvous, but also the server code for the **select**, thereby setting up for the next **entry** call. The client is still required to state switch to and from the server for each rendezvous.

3.4.3.3.3 Evaluation

In this section the performance of the two Habermann and Nassi proposals are compared with the performance of a normal implementation. The assumption is that the client task is higher priority than the server and therefore must run after completion of the rendezvous. The results of the client executing the rendezvous are analyzed first. For a single rendezvous, the following occurs: If the client arrives first, it is still blocked. A context switch transfers control to the server, which then executes down to the **accept** statement. At that point, the client is moved to the ready queue and a context switch is made to the client, who then executes the rendezvous code and then continues in its own code. Therefore, letting the client execute the rendezvous results in exactly the same number of context switches (two) as when the rendezvous is executed by the server. So, there are no savings in this situation. Next, consider the scenario of the server arriving at the **accept** first. When the server arrives at the **accept**, it is blocked. The client is then scheduled and executes down to the **entry** call. This requires a context switch. When the client arrives at the **entry** call, instead of blocking, it simply executes the rendezvous and continues in its own code after the **entry** call. No additional context switches are required. Therefore, instead of three context switches, as is the case with having the server execute a single rendezvous, only one is required. In summary, for a single rendezvous, letting the client task execute the server's rendezvous code saves two of the three context switches when the server arrives at the **accept** first, but saves none when the client arrives first. There is also some additional overhead involved in letting the client task execute the rendezvous. For one thing, some sort of guards must be used to insure that only mutually exclusive access to the server task is allowed. In the case of the Habermann and Nassi technique, this is done with semaphores. Also, some of the actions of the context switch are still required for the rendezvous. At first glance, it may appear that no switch of any kind is required in this situation. That is, however, not true. Registers and machine status may need to be saved during the execution of the rendezvous.

Furthermore, the existing referencing environment (scope) of the client must be saved, and it must be temporarily replaced with that of the server. When the rendezvous is complete, the client's environment must be reestablished. In essence, this alternative implementation replaces two context switches, from the client to the server and back, with two state switches. One of the positive aspects about the Habermann and Nassi work is that they concerned themselves with most of the details of the tasking model in their implementation. For example, they show how their procedure can handle **delay** alternatives, **else** parts, nested **accept** statements, and exception handling during rendezvous (which requires that the exception be handled by both the client and the server). After this examination, at least in the simple environment of execution considered here, it is concluded that there is little advantage to letting the client execute all rendezvous.

The second proposal of Habermann and Nassi, (to convert, when possible, server tasks to monitors) deserves close examination. First, recall that the method was not automatic. It was a hand modification, at least partially because they developed no automatic method of detecting situations where it could be done. Second, although there seems to be strong intuitive evidence that a monitor is consistently more efficient than a server task implemented by the normal method, that is not always the case. Eventoff, et al. (1980) do an extensive comparison of the relative efficiency of the monitor and the rendezvous. Their comparisons were done in a number of different situations. The most interesting cases at this point are those where contention was present for both producer and consumer tasks. Contention occurs when one or more tasks are blocked by the mutual exclusion mechanisms. In the presence of a high degree of contention, the study shows that, at least by the measures used in the study, the benefits of the monitor are reduced. This is because the monitor will be blocking both clients sending and receiving data. Another additional overhead of the monitor in situations of contention is that client tasks may be blocked twice before the

communication takes place, once at the mutual exclusion mechanism and once if the shared data structure is empty or full. There can be just one block of an Ada task before the communication takes place. The conclusion of this discussion is that although monitors can be very beneficial, their benefit is somewhat dependent on the application. There is a subtle form of serialization in the Habermann and Nassi monitor proposal, which forces all code to be executed as part of the rendezvous. Normally, the code after the **accept** statement is executed concurrently with the resumed caller. This has no effect on uni-processor machines, but can cause slightly slower execution on a multiprocessor or distributed processor system.

3.4.3.4 The Hilfinger Proposal

Hilfinger (1982) proposed a method of implementing collections of tasks called monitor clusters, which he claimed would result in significantly faster execution of these tasks than if they were implemented in the normal method.

3.4.3.4.1 Technique

Hilfinger defines a monitor cluster to be a collection of tasks with the following properties:

1. Tasks in the monitor cluster call only other tasks in the cluster.
2. There are no **delay** alternatives in tasks in the cluster.
3. There are no explicit priorities stated for any task in the cluster, but all cluster tasks run at a priority that is higher than any task in the program outside the cluster.

Calls from outside tasks are accepted only if there are no pending internal calls. Because all tasks in a monitor cluster run at the same priority, which is higher than that of any task outside the cluster, cluster tasks can control their own execution using a separate (from the RTS) and trivial scheduling process. In fact, a simple stack is used for storage of and scheduling of ready tasks in the cluster. Cluster tasks are executed as if they were coroutines,

thus replacing concurrency with serial execution. Full context switches are never necessary, although state changes are needed each time execution flows from one task to another. As is the case with most current implementations, execution stays in a task as long as possible. Execution flows from a client to a server by a simple jump instruction, and there are no automatic returns from such transfers. The client is put in the cluster ready queue before the jump, so its execution is resumed only when the cluster scheduler chooses it from the task ready queue. Note that while Hilfinger specifies a stack as the cluster ready queue, there is no reason a FIFO queue could not be used, which would result in a slightly higher level of "fairness" of scheduling of cluster tasks.

3.4.3.4.2 Evaluation

This method effectively removes all context switches from changes in control within the monitor cluster. Furthermore, the number of state switches is minimized because the number of scheduling points is minimized. The end result is that running tasks as coroutines should be faster than running them as tasks in a normal implementation, at least on a uni-processor system. Actually, the difference between the context switches of a normal implementation and the state switches of the cluster tasks is that the scheduling is simplified. So that is the source of the time savings. The Hilfinger proposal effectively serializes execution of a supposedly concurrent program. The questions naturally arise: Are there useful applications of such a restricted form of tasking? Presumably, such a program could have been developed as a set of procedures since there appears to be no benefit in using tasks for such an application. Obviously, this approach would need substantial modification for use on a multiprocessor.

3.4.3.5 The Shauer Proposal

Shauer (1982) proposed a method of increasing the speed of rendezvous that is closely related to the second proposal of Habermann and Nassi.

3.4.3.5.1 The Technique

The Shauer proposal is to effectively convert the **accept** statements of server tasks to subprograms. If this could be done, it would reduce all context switches to state switches, and also remove the need for the cactus stack, because subprogram activations can run on a single runtime stack. The following description of Shauer's work is taken from Burns (1985), because Shauer's paper was inaccessible. The Shauer implementation of a server task requires the following conversions:

1. Move all extended **accept** statement code into the preceding **accept** statement and structure each **accept** statement as a procedure.
2. Encapsulate all code preceding the first **accept** statement in a special "start" procedure.
3. Replace each guard with a conditional WAIT operation on a *semaphore, which* includes a queue for storing suspended callers.
4. Treat all **entry** calls to the converted server as procedure calls.
5. Replace the server task initialization by a call to the start procedure.

The whole server task could be converted to a package, because having it remain a task seems to serve no purpose. Then the initialization code could be placed in the body of the package. Mechanisms to provide mutually exclusive access to the subprograms would, of course, still be necessary, as would code to handle selection choice. It may be necessary to restrict the servers on which this transformation could be done to those without **delay** and/or **terminate** alternatives. Nested **accept** statements may also be difficult to handle. Calls to other tasks from the package, however, should pose no problems.

3.4.3.5.2 Evaluation

Shauer's technique is very similar to the second Habermann and Nassi proposal. It is a more complete serialization because it applies to servers capable of a higher level of concurrency (those with extended **accept** statements). Therefore, a Shauer implementation would be a greater deterrent to physical concurrency than the Habermann and Nassi technique. On uni-processor systems, however, it would be an efficient method of implementation.

3.4.3.6 The Stevenson Proposals

Stevenson (1980) proposed several ideas for efficient implementations of Ada tasks.

3.4.3.6.1 The Techniques

Stevenson suggested that a variety of methods of implementing Ada tasking be tried and compared. To do this, a target language was designed that would clearly illustrate the various algorithms for implementation. This target language, named Ada-M, is similar to Ada in its sequential parts, but uses a lower-level method of specifying concurrency. Much of Stevenson's paper describes how the translation to Ada-M could be done, using one of the two proposed implementation techniques as an example. The focus here is on the implementation methods, and Ada-M is largely ignored. The first implementation method proposed by Stevenson is called the "procedure call" method. This is essentially the same as the first technique proposed by Habermann and Nassi. The client always executes the rendezvous code, but the server still executes the select control code. Stevenson's second proposed implementation method is called "order of arrival," and its name exactly describes its technique. The basis for the method is that the context switch at the beginning of a rendezvous can always be avoided by letting the last task to arrive execute the rendezvous. Stevenson does not specify whether control shifts to the caller after completion of the

rendezvous, although there is little justification for that. It makes more sense to leave control in a task as long as possible.

3.4.3.6.2 Evaluation

The result of this approach is that each rendezvous requires only a single context switch (assuming equal priorities). This is half as many as any of the other methods we have investigated here. Every other rendezvous, specifically those that are executed by the caller, require two state switches. So, this method saves one context switch per rendezvous, at a cost of an average of one state switch per rendezvous. Stevenson's order-of-arrival method is clearly the best software method we have found to implement rendezvous when serialization is undesirable.

3.4.4 Other Optimizations

Several other optimizations of task implementation have been suggested, primarily by Frankel (1987). In the following, we explain and evaluate these.

3.4.4.1 **accept** Statements Without Bodies

An **accept** statement that has no body does not require a state switch at the point of the rendezvous. The rendezvous, in this case, is merely a signal from one task to another, stating that execution has reached the point of interest (as marked by the **entry** call in the client and the **accept** statement in the server). Such **accept** statements should require very little overhead. If the client arrives first, it is blocked. When the server arrives, the client must be moved to the ready queue. If the server arrives first, it is blocked. When the client arrives, the server must be moved to the ready queue. It appears to be a simple matter to have the compiler recognize **accept** statements without bodies and avoid generating code to do

anything except the blocking and unblocking of the first task to arrive at such an accept. Obviously when placing a task on the ready queue, priorities must be observed and preemption may occur.

3.4.4.2 Asynchronous Messengers

An asynchronous messenger task is an intermediate, or agent, task whose only job is to transfer messages one way to a server task, without forcing the client to wait for a response from the server. Their actions explain their purpose: They are meant to allow a client to asynchronously send a message to a server task. (A synchronous message is one that requires the client to wait for a reply from the server before continuing. An asynchronous message is one that allows the client to continue its execution as soon as its sends the message.) The parameters of an asynchronous message must be in mode only, because no information can flow backwards through such a message if the client is free to leave the entry call immediately. Asynchronous messages using messenger tasks are expensive because one rendezvous is replaced by two rendezvous. Given the high cost of even a simple rendezvous, any optimization of this action would be helpful. The somewhat obvious optimization would be to have the compiler simply remove the agent task, replacing it with a message queue attached to the server. The client could place its messages in the message queue whenever it wishes, and the server could accept the messages whenever it wishes. The only problem with this optimization is that it would be very difficult for the compiler to recognize situations in which the optimization could be applied. User programs can use different forms of such agent tasks (Burns, 1985), which need not appear in any standard, easy to recognize form (e.g., they may or may not be encapsulated in packages). One solution to this problem is for the implementation to provide a pragma that the user could use to indicate when a task is an agent that could be optimized away. That would allow the user to specify legal Ada code, but still allow an efficient implementation.

3.4.4.3 Replacement of Entry Families Used for Prioritized Calls

Families of entries can be used to achieve the effect of prioritized calls on a single entry. The different family member entries are associated with different priorities, so that calls with different indices refer to entries with different priorities. Burns (1987) recommends two different methods of using entry families to specify prioritized calls. For situations requiring a relatively small number of different priorities, the COUNT attribute of entries is used in guards of accept statements to control access. Entries with low priority are guarded until there are no waiting messages at any higher priority entries. When the number of different priorities is large, a different technique must be used, because the guards quickly become excessively large, making them tedious to write and costly to repeatedly evaluate. An alternative method for this situation, as advocated by Burns, is more efficient, but also more complex. This technique requires a client to call twice to complete the message transmission. The first call is used to announce that the caller wants to communicate and also to deliver the priority of the imminent message. The second call actually sends the message to the server task. There is significant overhead to this process. For one thing, it requires twice as many rendezvous. It also requires another agent task, which manages a storage structure to store announcement calls. The obvious optimization for an entry family used to achieve priority order in an entry queue is simply to convert it to what the user wanted to begin with: a single prioritized queue on a single entry. Although this would certainly result in a significant improvement in performance, it is difficult to see how the circumstances under which this conversion is applicable could be recognized by the compiler. While the Burns suggestions above are good ones, there is no reason to believe all Ada programs will use those techniques exactly. Therefore, there is simply no standard way of using entry families for these purposes that would allow easy compiler recognition.

3.4.5 Architecture

All of the approaches to increasing the speed of Ada task interaction that have been discussed thus far have dealt with software. In this section machine architectures that have the potential for increasing the speed of task execution and interaction are investigated. In the following subsections, two architectural approaches, special instructions and coprocessors are discussed.

3.4.5.1 Special Instructions

Complex but often used operations can be implemented in three distinct ways. They can be constructed as hardware instructions, using electronic circuitry, which makes them very fast but very expensive to construct. Floating point arithmetic operations on larger machines, for example, are implemented with hardware. Such operations can also be coded as machine instruction sequences (software), which makes them far less expensive to build, but much slower. Floating point arithmetic operations on very small machines are often implemented this way. The third method, firmware, is a compromise between hardware and software, with the result being between the other two, in both cost of construction and in execution speed. (Firmware is similar to software, except that microcode instructions are used instead of machine instructions, and the microcode resides, permanently, in read-only memory. Microcode is a lower level language than machine language.) Many relatively inexpensive computers of the 1970s used firmware to implement floating point arithmetic operations. Likewise, many of the most complex instructions of contemporary computers are implemented in firmware. Considering the cost of hardware, firmware would appear to be a good method to provide instructions specifically to support Ada task interaction. We discuss research in the next section that includes special hardware instructions for tasking primitives, but in the context of a coprocessor environment. That work defines a set of such primitives that would be the best candidates for firmware implementation. Firmware implementations

of task interaction processes would be significantly faster than software, which is now used in the majority of implementations. The Rational R1000 computer system, which was designed specifically to implement Ada efficiently, uses some special instructions in firmware for task interaction. Unfortunately, because the design of this system is proprietary, it is difficult to determine the extent and impact of its use of firmware. There have been several efforts to design processors (in hardware) that were general in nature, but more supportive of Ada in particular (Nokia, 1983) (Biswas and Dasgupta, 1985) (Ibsen, et al, 1983). As reported in Roos (1989), these had the goal of improving the average performance of an entire Ada program. However, these particular efforts produced little speedup of rendezvous.

3.4.5.2 Coprocessors

The idea of adding a coprocessor to a computer to support Ada tasking is relatively obvious. There are, however, several ways in which such a coprocessor could be used. As in every other environment, one would like to achieve the highest performance while requiring the minimum cost and complexity.

3.4.5.2.1 Tasking Coprocessor Method

The first, and the most important, coprocessor usage method discussed is the tasking coprocessor method. It is most important because it is perhaps the most effective method, and also because it has been actively investigated and reported in the literature, whereas the other methods discussed here are only preliminary ideas. In the following, the work done at Lund University in Sweden is described, as reported by Roos (1989). The approach of the tasking coprocessor method is as follows: A special purpose processor is designed specifically to execute primitive operations that implement task scheduling and interaction. It is therefore a coprocessor implementation of special instructions, as discussed briefly in

Section 3.4.5.1. The coprocessor used (in the Lund project) is a 2 micron CMOS chip with a 20 MHz clock rate, which results in a typical operation time of about 1 microsecond. Teamed with a 1 mip CPU, the tasking primitives are done in one or just a few CPU instruction cycles. The coprocessor's relationship with the CPU is similar to that of a floating point coprocessor: When a tasking primitive operation is required, the CPU sends commands and the required operands to the coprocessor, which executes them and the CPU reads the results. The coprocessor has on chip RAM that is used to store task control blocks and queue headers. Three kinds of queues are maintained: entry, delay, and ready. Because these queues reside on the coprocessor chip RAM, only coprocessor operations can access them. This implicitly ensures mutually exclusive access. Communication between the CPU and the coprocessor is based on memory-mapped read and write instructions. The set of operations for the coprocessor were determined by use of three criteria:

1. The operation must require a significant amount of code.
2. The amount of data communication between the CPU and the coprocessor must be small.
3.It must be possible to implement the operation efficiently using current VLSI technology.

A sampling of the operations implemented on the coprocessor include the following:

- CreateTask,
- ActTask (activate task),
- TimedECall (make a timed or conditional call),
- SelectRes (choose an alternative),
- Switch (perform a scheduling),
- Terminated (T'terminated),
- and Suspend.

Performance of the coprocessor was checked using a simulator provided with the tool used to design the coprocessor. The results are very impressive. A simple rendezvous requires 8.4 microseconds, which is sixty times faster than when run in software on a VAX8600 (see Section 3.4.1). The most complex rendezvous investigated, a timed call to a selective wait

with two **accept** alternatives and a **terminate** alternative, took only 14.8 microseconds. Because of the dramatic speed increase that results from the use of this method, it must be considered the most promising technique for using a coprocessor to implement tasking. The most serious concern is the very high cost of a custom VLSI circuit considering the low volume that would be expected.

3.4.5.2.2 Twin Processor Method

A coprocessor that was identical to the CPU could be added as a second, equal processor. This is called a twin processor method. The two processors would be treated as equals and scheduled accordingly. This would clearly provide a boost in overall performance to programs that contained multiple active tasks, because it would, at least theoretically, double the capacity of the system for physical concurrency. With a normal implementation of tasking, programs with a single active task would, of course, be only lightly affected by the coprocessor. The costs of a twin processor system include both hardware and software needs. Contention for memory, which is shared by the two processors, is usually not a severe problem with only two processors, because each uses less than a fraction of the available bandwidth to memory. If the two processors have cache memories, which is becoming more and more likely with advances in technology, then additional hardware is required to maintain the integrity of cache contents. When one processor changes the contents of a memory location that is currently resident in the other processor's cache, that cache location must be either invalidated or implicitly reloaded from memory.

3.4.5.2.3 Background Coprocessor Method

One more use for a coprocessor in a system meant to run Ada programs is the following. A coprocessor could be used to do some background kinds of processing, concurrently with the

CPU (or multiple CPUs), which could be multiple. The background processing could include maintaining the ready queue. When explicit priorities are specified for tasks, the ready queue must always be ordered by priority. All of the ordering process could be off-loaded from the CPU. When the RTS schedules a task, it can simply take it off the end of the sorted list of tasks maintained by the coprocessor. Maintenance of the master list is another candidate for background processing. In this case, the coprocessor could be simpler and slower, and therefore cheaper than the CPU. It could have a small memory of its own, in which it could store the data structures upon which it operates.

3.5 Best Utilization of Ada Tasking Model for Uni-processor Applications

This section assumes that the target system will only have a single processor per Ada program. For these systems, the main benefit of tasking is to allow the continued processing of other activities while waiting for inputs. Although this can be done using polling in a non-tasking solution, often the overhead and interference of polling to the application is so severe that it is not practical. The general recommendation is to avoid unnecessary use of tasking. Cases where activities can be serialized should be handled using procedures rather than tasks. Special effort should be taken to avoid reliance on "abort" or dynamic task creation. Analyze the interaction of tasks and determine the number of context switches that could occur. If the context switch rate is excessive, measures should be taken to lower the rate. In some cases, careful use of shared variables can reduce the need for rendezvous, but these must be thoroughly studied since they are frequently the source of programming errors.

These recommendations should be considered as guidelines only, and with an understanding that there are cases where tasking is desirable. Tasks which service I/O requests, particularly interrupts, are extremely useful because they provide fast response to sporadic

events. Of course, the efficient use of Ada interrupt entries depends on the implementation's support for optimized interrupt handling. Although task abortion and creation are time consuming activities, there may be some good uses of these features. Any use of these features should be justified as reasonable in terms of the application requirements and the available CPU utilization. Performance of Ada tasking is adequate if these guidelines are followed along with the use of a high performance runtime and discretionary use of task interaction.

3.6 Best Utilization of Ada Tasking Model for Distributed Applications

Multiprocessor use of tasking increases the potential to take advantage of the concurrency within applications. For this reason, systems that are candidates for multiprocessors where the Ada tasks can be allocated to different CPUs should take a different approach then the one described for uni-processors. In particular, the natural concurrency in an application should be expressed in the form of tasks. Where the concurrency is great (e.g. target tracking) the use of task arrays should be used with the task type being able to handle a configurable number of operations per rendezvous. This allows the program to be configured so as to expand the number of tasks and to take advantage of the available CPUS. When the number of available CPUs is small, then only a small number of tracking tasks need be configured and the number of rendezvous will be kept low. This is because the workload is distributed to the available tasks as lists of operations rather than as a single operation per rendezvous. Although this complicates the design somewhat, the benefits of increasing throughput outweigh the additional complexity. The communication to task arrays does have the unfortunate side effect of forcing a particular order on the rendezvous. This could cause substantial delays in cases where the individual tasks in the array are not available at nearly the same time. In these cases, conditioned calls may be appropriate until

all of the tasks have rendezvous. However, the cost of the additional conditional calls may also be high in a distributed system. The tradeoffs of this approach must be evaluated for the typical application behavior.

4. Summary of Results, Recommendations and Conclusions

Proposals for increasing the speed of implementations of task scheduling and interaction have been examined and reported. These include suggestions that all rendezvous be executed by the calling task, that all rendezvous be executed by the last task to arrive, that server tasks be converted to monitors, that tasks in a cluster monitor be run as coroutines, and that servers be effectively converted to packages of guarded subprograms that provide the actions of the rendezvous of the server. The software optimizations fall into two distinct categories: those that serialize task execution and those that do not. Serialization is a valid means of increasing performance on uni-processor systems, but because it limits or disallows physical concurrency, it has a negative impact on the performance of multiprocessor systems. Note that the serialization came as a by-product of the action of bypassing the complexity of the Ada tasking model. Complex operations are replaced by lower-level operations; rendezvous by subprogram call and semaphores. The best serialization technique is the use of monitors which provide substantial benefits for applications where contention is likely to be low. When serialization is undesirable, Stevenson's order-of-arrival method is clearly the most efficient software method of implementing rendezvous, at least in the execution environment in which the methods were evaluated.

The Ada tasking model is complex and defined at a high level of abstraction. The price of this is that implementations that are complete are going to be relatively slow. Concurrency can be far more efficient when the mechanisms are primitive, as in the case of semaphores. The price of this primitiveness is that the primitives are easier to misuse. The key question is whether the user community is willing to trade the decrease in performance for the ease of use. It is clear that a concise analysis of the costs and benefits of the software optimization techniques investigated in this report would be difficult and costly. In some cases a simulation could be used. In others, prototype implementations would be more appropriate.

Ways in which a coprocessor can be added to a system and used to increase the performance of rendezvous were discussed. The tasking coprocessor is clearly the most technically appealing of these, both because it so effectively reduces rendezvous time and because it is the most studied method. The best alternative to the tasking coprocessor system is the background coprocessor system, because of its simplicity and projected low cost, in spite of its potential significant positive impact on performance. Any use of a coprocessor carries with it the negative of disallowing off-the-shelf processing systems, as many embedded systems are now configured. In most cases, the use of coprocessors require the modification of the compiler and/or RTS. This is not a simple or inexpensive matter in the case of Ada.

The overall conclusion is that Ada tasking performance is unlikely to improve substantially beyond what is currently available in the high performance runtimes unless hardware support is provided. Most of the optimizations reported publicly have been implemented to some degree in these runtimes which are written in assembly language. The remaining 150 microseconds involved in an actual rendezvous on a 25MHz 80386 is the result of the complexity of Ada and is unlikely to be eliminated without hardware support. Many questions remain as to the commercial viability of specialized hardware support due to the limited market for its use. This question is the focus of the experiment proposed in the next section.

5. Further Experiments

The conclusions of this study indicate that little or no performance enhancements are likely to be forthcoming in Ada tasking without special hardware support. The Ada tasking model is very complex and requires certain activities to be done during task communication. Some runtime vendors have been able to achieve relatively high performance runtimes (rendezvous under 200us) for simple cases and provide additional synchronization mechanisms (semaphores) for applications where this is too slow. These performances have been achieved largely by hand optimization of assembly language runtimes. Other optimizations, such as fast interrupt pragmas take advantage of special cases where the need to use the runtime can be substantially reduced. Although several runtime vendors still have much lower performance tasking, this is largely due to the lack of emphasis placed on optimization. To some degree the marketplace has been biased towards sequential programs for non-embedded applications which is largely unconcerned with the performance of a context switch.

The hardware approach can take on several forms. Obviously, any enhancement to speed up the average performance of a processor (such as Reduced Instruction Set Computer (RISC) technology) will benefit tasking services as well. Other than this, the highest performance and probably least flexible approach would be to incorporate full Ada tasking semantics in hardware as part of a standard microprocessor. The closest example of this approach is the Intel 80960MC architecture which incorporates a prioritized scheduler and delay services in on-chip microcode. However, this is a small portion of supporting the Ada semantics and therefore the expected improvement is small.

Another approach is to have a custom hardware chip developed to provide tasking primitives. The University of Lund work described in this report is an example of this

technique. The major problem with this approach is cost. It now appears that it is not commercially viable for such an approach to succeed. Integrated Circuits (ICs) typically sell in quantities of 1000's to 100,000 per month. A typical Ada compiler for embedded use normally sells at the rate of a few per month. The complexity of the IC (and therefore the IC size) insures that the price will never reach commodity prices (under $100/IC). This precludes large volume applications. Instead, the likely market is low volume, extremely high performance systems. However, the benefit of the IC is relatively small as a total part of CPU processing, and therefore adding another CPU is a much more cost effective solution than going to a custom IC.

Finally, an approach which could be tried is the use of an extremely low cost (under $10 in volume) single chip MicroProcessor Unit (MPU) as a background coprocessor. This approach would have the tasking services and data structures maintained on-chip as in the custom IC approach, but would provide the customizing via software. The major question regarding this approach is the degree to which the tasking services can be parallelized with respect to Ada tasking semantics. For example, it must be possible to issue a "create task" request and then immediately issue a "select next task to run" request. If the MPU has to execute all of the instructions in the create task primitive prior to servicing the scheduling request, it would be better to have the main CPU perform the operations since it is likely to be considerably faster. However, the presumption is that much of the MPU work can be overlapped and therefore response to nearly all CPU requests will be prompt, with the MPU doing the work at its earliest convenience. Obvious areas of contention are with task creation, but this is probably not a serious problem due to the low frequency of use. The primary emphasis should be with support for the rendezvous and timing services.

This experiment will design and implement the tasking support for an Ada runtime using the MPU coprocessor approach. An equivalent instruction set to the one implemented by the University of Lund coprocessor shall be used. A prototype system will be designed using memory mapped access to the MPU. Instruction counts for each of the services listed in the taxonomy will be collected as well as PIWG performance benchmarks. These will be a "paper count" generated by counting the instructions necessary to implement the MPU functions. That is, no hardware need be developed, however the software must be produced to support the tasking primitives. The times required to execute the MPU instructions should be characterized for the parameters so that benchmarks can be evaluated.

The critical aspect of this experiment is the design of the background coprocessor software. The most sensitive area is the degree to which the results required by the main processor can be "precomputed" and quickly provided when needed. To some degree, the order in which the main processor issues requests can be optimized to increase the overlap. For example, the main CPU may issue a suspension request for the current task to the MPU, save its own registers, then request the next task to run from the MPU. The time used to save its registers may only be a few microseconds, but it is completely overlapped with the MPU execution.

Another real benefit is the ability to provide extremely accurate delays without the overhead of frequent interrupts or complex timer manipulation. This can be very important for many applications. For example, to achieve 1ms resolution often requires in excess of 10% overhead to process an interrupt every millisecond and determine which, if any, tasks must be resumed. Since Calendar.CLOCK is also updated during these interrupts, the interrupt overhead and service routine can easily exceed 100us resulting in a loss of 10% of the CPU utilization. This is true even if the delay that needs the 1ms resolution is executed very infrequently.

These issues will be investigated and a report describing the benefits and limitations of the resulting implementation, as well as difficulties of the project shall be produced.

6. References

[1] Biswas, P. and S. Dasgupta (1985). Architectural Support for Variable Addressing in Ada - A Design Approach. Journal of Computer and Information Sciences, Vol. 14, No. 1, pp. 51-72.

[2] Brinch Hansen, P. (1973). Concurrent programming concepts. ACM Computer Surveys, Vol. 5, No. 4, pp. 223-245.

[3] Brinch Hansen, P. (1975). The programming language Concurrent Pascal. IEEE Trans. on Software Eng., Vol. 2, pp. 199-207.

[4] Brinch Hansen, P. (1978). Distributed processes: A concurrent programming concept. Commun. ACM, Vol. 21, No. 11, pp. 934-941.

[5] Burger, T.M. and K.W. Nielsen (1987). An assessment of the overhead associated with tasking facilities and task paradigms in Ada. ACM Ada Letters, Vol. 7, No. 1, pp. 49-58.

[6] Burns, A. (1985). Concurrent programming in Ada. Cambridge Univ. Press, Cambridge.

[7] Burns, A. (1987). Using large families for handling priority requests. ACM Ada Letters, Vol. 7, No. 1, pp. 97-104.

[8] Clapp, R.M., L. Duchesneau, R.A. Volz, T.N. Mudge, and T. Schultze (1986). Toward real-time performance benchmarks for Ada. Commun. ACM, Vol. 29, No. 8, pp. 760-778.

[9] Dijkstra, E.W. (1968). Cooperating sequential processes. Programming Languages, ed. F. Genuys, Academic Press, New York.

[10] Dijkstra, E.W. (1975). Guarded commands, nondeterminancy and formal derivation of programs. Commun. ACM, Vol. 18, No. 8, pp. 453-457.

[11] Eventoff, W., D. Harvey, and R.J. Rice (1980). The rendezvous and monitor concepts: Is there an efficiency difference? ACM SIGPLAN Symp. on Ada, Boston, Nov. 1980, pp. 156-165.

[12] Frankel, G. (1987). Improving Ada tasking performance. Proc. ACM International Workshop on Real-Time Ada Issues, Ada Letters, Vol. 8, No. 7, pp. 47-48.

[13] Habermann, A.N. and I.R. Nassi (1980). Efficient implementation of Ada tasks. Technical Report CMU-CS-80-103, Dept. of Computer Science, Carnegie-Mellon Univ., January 1980.

[14] Hilfinger, P.N. (1982). Implementation strategies for Ada tasking idioms. ACM AdaTEC Conf. on Ada, Arlington, Va., Oct. 1982, pp. 26-30.

[15] Hoare, C.A.R. (1978). Communicating sequential processes. Commun. ACM, Vol. 21, No. 8, pp. 666-677.

[16] Ibsen, L., L.O.K. Nielsen, and N.M. Jorgensen (1983). A-machine Specification. Christian Rovsing A/S, Denmark, Document ADA/RFM/0001, issue 2.

[17] Ichbiah, J.D., J.G.P. Barnes, J.C. Heliard, B. Krieg-Brueckner, O. Roubine, and B.A. Wichmann (1979). Rationale for the design of the Ada programming language. ACM SIGPLAN Notices, Vol. 14, No. 6, June 1979, PartB.

[18] Nokia Data (1983). MPS-10 Documentation Series. Nokia Data, Terminal Systems, Finland.

[19] Roos, J. (1989). A Real-Time Support Processor for Ada Tasking. ACM SIGPLAN Notices, Vol. 24, Special Issue, May, pp. 162-171.

[20] Shauer, J. (1982). Vereinfachung von prozess--Systemen durch sequentialisievung. 30/82, Institut for Informatik, Bericht.

[21] Stevenson, D.R. (1980). Algorithms for translating Ada multitasking. ACM SIGPLAN Symp. on Ada, Boston, Nov. 1980, pp. 166-175.

[22] CECOM, Center for Software Engineering, Advanced Software Technology, "Real-Time Demonstation Project", CIN: C02 092LA 0009 00, prepared by LabTek Corp., 27 April 1989.

[23] T. P. Baker and G. A. Riccardi, "Ada Tasking: From Semantics to Efficient Implementation," IEEE Software, Volume 2, Number 2, March 1985, pp. 34-46.

[24] G. A. Riccardi and T. P. Baker, "A Runtime Supervisor to Support Ada Task Activation, Execution and Termination (preliminary report), "Proceedings of the IEEE Computer Society 1984 Conference on Ada Application and Environments, October 1984, pp. 14-22.

[25] G. A. Riccardi and T.P. Baker, "A Runtime Supervisor to Support Ada Tasking: Rendezvous and Delays, "Proceedings of the Ada International Conference 1985, pp. 329-342.

[26] ANSI/MIL-STD-1815A-1983. "Reference Manual for the Ada Programming Language", American National Standards Institute, Inc., 1983.

[27] CECOM, Center for Software Engineering, Advanced Software Technology, "Final Report - Real-Time Performance Benchmarks for Ada", CIN: C02 092LY 0001 00, 24 March 1989.

7. Appendix A: The Ada Tasking Model

Ada is the first widely used programming language to implement message passing as a method of providing support for concurrency. This design and its implementation is, therefore, somewhat of an experiment. In the following we briefly describe the Ada model of message passing. Note that this description of Ada tasking facilities is not meant to be a programmer's guide; rather, it is intended to inform the reader from the implementer's point of view. For a complete semantic definition of the Ada tasking model refer to Chapter 9 of the Reference Manual for the Ada Programming Language [26].

7.1 Basic Task Structure

An Ada task is a program unit that consists of two parts, the specification and the body. The specification usually contains only a single kind of declaration, which is used to describe the form of message sockets the corresponding task body contains. In the specification these are called entry statements. Task bodies contain declarations of local variables and a statement sequence that can include **accept** statements, which correspond to the entry declarations in the associated task specification. The **accept** statements describe the entry points for incoming messages and also the sequence of statements that are to be executed when a message is accepted. The general form of an **accept** statement is

```
accept entry_name [entry_index] [(formal_parameters)] [do
  sequence of statements
end [entry_name];
```

The bracketed items are optional. An **accept** statement can contain almost any other statements, including calls to other tasks and nested **accept** statements. The statement sequence of an **accept** statement defines the actions of the rendezvous. Rendezvous begins when the message is passed into the **accept** statement and execution of the statement

sequence begins. It terminates when the end of the **accept** statement is reached. Each **entry** has an associated queue, which is used to store pending **entry** calls that have been made but not yet accepted. The **accept** body, consisting of the reserved word **do**, the statement sequence, and **end**, is optional. An **accept** statement without a body is used for synchronization, without any data communication or rendezvous actions.

7.2 Selective Wait

Sometimes an **accept** statement is used repeatedly, and is placed in a loop construct. In many cases, multiple **accept** statements are collected into a structure and used repeatedly. The syntactic structure for collections of related **accept** statements is a form of the **select** statement called a selective wait, which can have the form:

```
select
  accept ...
or
  accept ...
or
  ...
end select;
```

The selective wait is usually placed in a loop structure, which causes its repeated execution. The semantics of a simple selective wait, such as the one above, is that each time the beginning of the **select** is reached, the entry queues referenced in each of the **accept** alternatives are examined. If exactly one of the queues is nonempty, the next message in that queue is chosen for acceptance. This causes the statement sequence in the associated **accept** alternative to be executed. When the rendezvous is completed, the **select** statement is exited. If more than one **accept** alternative have nonempty entry queues, one of them is chosen, according to the implementation defined approach. If none of the **accept** statements have a nonempty queue, execution is suspended until a message arrives. Note that messages in a given entry queue are always accepted in FIFO order.

7.3 Guarded accept Alternatives

Accept alternatives in **select** statements can have guards, in the form of Boolean expressions, to limit their ability to accept messages. The general form of a guard is:

when Boolean_expression = > **accept** ...

When the Boolean expression in a guard is true, the **accept** alternative is said to be open; otherwise it is closed. An **accept** alternative without a guard is always open. An open **accept** alternative can rendezvous; a closed **accept** alternative cannot. A **select** statement with one or more guarded **accept** alternatives has the following semantics. When control reaches the **select**, all guards are evaluated and a list of open **accept** alternative is formed. If none of the **accept** alternatives is open, the PROGRAM_ERROR exception is raised (see **else** below). If more than one open **accept** alternative has a nonempty queue, one of them is nondeterministically chosen for message acceptance. If exactly one open **accept** statement has a nonempty queue, the next message in its queue is chosen for acceptance. If none of the open **accept** statements has a nonempty queue, execution is suspended until a message arrives at one of the open **accept** alternatives.

7.4 else Parts

To continue execution when there are no open **accept** alternatives, or when all the open alternatives have no corresponding **entry** calls, an **else** part may be included at the end of the **select**. The **else** part is executed when the **select** is executed and all of its open **accept** alternatives have empty queues. The **else** part allows the task to carry out some background computation when no **entry** calls which can be accepted are waiting.

7.5 A Shared Buffer: An Example

A task body that controls access to a shared buffer that stores INTEGER type values and provides INSERT and REMOVE operations is shown below.

```
task BUF_TASK is
  BUF_SIZE : constant INTEGER := 100;
  BUF      : array (1..BUF_SIZE) of INTEGER;
  FILLED   : INTEGER range 0..BUF_SIZE := 0;
  NEXT_IN,
  NEXT_OUT : INTEGER range 1..BUF_SIZE := 1
  begin
    loop
      select
        when FILLED < BUF_SIZE =>
          accept INSERT (ITEM : in INTEGER) do
            BUF (NEXT_IN) := ITEM;
            NEXT_IN := (NEXT_IN mod BUF_SIZE) + 1;
            FILLED := FILLED + 1;
          end INSERT;
      or
        when FILLED > 0 =>
          accept REMOVE (ITEM : out INTEGER) do
            ITEM := BUF (NEXT_OUT);
            NEXT_OUT := (NEXT_OUT mod BUF_SIZE) + 1;
            FILLED := FILLED - 1;
          end REMOVE;
      end select;
    end loop;
  end BUF_TASK;
```

Producer and consumer tasks can use the buffer controlled by this task by calling its operations when needed.

7.6 Rendezvous

A message send operation is called an entry call, and in its simplest form has syntax that is similar to a procedure call. Entry calls name their intended entries with the names of the called task and the entry, catenated with a period. For example, BUF_TASK.INSERT is used to call the INSERT entry of BUF_TASK. We describe the more complex form of entry calls later in this section. The semantics of rendezvous is that the calling task is blocked (suspended) from the time the message is sent until after the last statement of the accept

statement is executed. At that time the caller is placed back in the ready queue, which makes it available for execution.

7.7 Extended accept Alternatives

When it is desirable to increase the level of concurrency of a program, it is possible to move the statements at the end of the **accept** statement that do not reference the parameters to a position below the end of the **accept**, but before the following **or**. This will allow the client task will be placed in the ready queue sooner, thus potentially increasing the level of concurrency. As an example, consider the modified version of the first part of the **select** statement of the task BUF_TASK:

```
...
when FILLED < BUF_SIZE =>
  accept INSERT (ITEM : in INTEGER) do
    BUF (NEXT_IN) := ITEM;
  end INSERT;
  NEXT_IN := (NEXT_IN mod BUF_SIZE) + 1;
  FILLED := FILLED + 1;
or
  ...
```

The code after an **accept** alternative that appears before the following or in a **select** statement is called an extended **accept** alternative. As stated above, the code in an extended **accept** alternative can be executed concurrently with the calling task. Another aspect of Ada semantics of tasks is that only one **accept** alternative can be active at a given time. This provides implicit mutual exclusion for the buffer task above. This part of task semantics is taken directly from the design of a monitor. There are several more complications that can occur with the **select** statement.

7.8 terminate Alternatives

Briefly, an explanation of the initiation and termination of tasks is necessary. Tasks that are created by declarations begin execution when the specification part in which they are declared is completely elaborated. As is discussed in Section 7.15, tasks can also be dynamically created using allocators. Such tasks begin their execution at the time they are allocated. Every Ada task has a master, which is the block, subprogram library package, or task whose execution created the task object,or in the case of allocated tasks, the master that declared the access type definition. A task is said to depend on its master. The importance of this is that the master cannot conclude its execution until all of its dependent tasks have terminated. A task can terminate simply by reaching the end of its statement sequence. Another possible way to terminate is to reach a **terminate** alternative in a **select** statement. But the only way a **terminate** alternative can be selected for execution is if the master and all of its other dependent tasks have either terminated or are waiting in a **select** statement that contains a **terminate** alternative. The task BUF_TASK could use a **terminate** alternative, which would allow it to terminate if its master and all of its client tasks had either terminated or were waiting at **terminate** alternatives. BUF_TASK, as written earlier in this section, does not terminate until the main program terminates.

7.9 delay Alternatives

The **delay** statement in Ada has the form:

```
delay expression;
```

The expression specifies a value that is interpreted to be a measure of time in seconds. The semantics are that the **delay** statement causes the task to be suspended for at least the amount of time specified in the expression. **Delay** statements can also appear in select wait alternatives, as a delay alternative, taking the place of **accept** alternatives. The semantics of

such a **delay** are different than for the same statement outside a **select**. Consider the following example.

```
select
  accept WHATEVER;
or
  delay 10.0;
end select;
```

The semantics of this **select** is that if no **entry** call is queued for WHATEVER, then a clock is started. If no message appears for the entry WHATEVER by the time the select clock reaches 10 seconds, the **delay** alternative is selected, which in this case causes the **select** to be exited. This provides a means of specifying that if a message is not received within a time limit, the task should stop waiting for one. A **delay** alternative can be followed by a statement sequence. These statements are executed when the **delay** is selected. As is described in Section 7.11, **delay** has another use in task interaction.

7.10 Conditional entry Calls

There are three different forms of entry calls. The simplest of these has the form of a procedure call. The other two use special forms of the **select** statement to specify different semantics.

The conditional entry call has the form:

```
select
  entry_call_stmt;
  [sequence_of_statements]
else
  sequence_of_statements
end select;
```

Note that the square brackets above mean that what is enclosed is optional. The semantics of the conditional entry call is as follows: An attempt is made to call the specified entry. If

the associated task can immediately accept, the rendezvous takes place. If the rendezvous cannot take place immediately, the attempt at rendezvous is abandoned and the sequence of statements in the else part is executed.

7.11 Timed entry Calls

The general form of the timed entry call is

```
select
  entry_name [(actual_parameters)];
  [sequence_of_statements]
or
  delay expression;
  [sequence_of_statements]
end select;
```

The semantics of the timed entry call are as follows: A timer is started when the select is reached. A rendezvous with the specified entry is attempted, and if immediate rendezvous is possible, it is carried out. If immediate rendezvous is not possible, then the calling task waits until either the select timer reaches the value specified in the delay statement or rendezvous occurs. If the timer runs out before the rendezvous occurs, the sequence of statements immediately following the delay statement is executed and any attempts at rendezvous with the specified entry are abandoned. If the rendezvous occurs, the sequence of statements following the entry call is executed after completion of the rendezvous.

7.12 entry Families

Entries can appear in families. For example, we could have a task specification such as:

```
task DO_IT is
  entry WHATEVER (1..5) (PARAM : INTEGER);
end DO_IT;
```

DO_IT is a task with five entries, named WHATEVER (1), WHATEVER (2), and so forth. The body of DO_IT may contain an **accept** statement for any of the five entries, and they may be all represented by a single **accept** statement form. For example, DO_IT's body could take either of the two following forms:

```
task body DO_IT is
  ...
  accept WHATEVER (1) (PARM) do
    ...
  end WHATEVER (1);
  accept WHATEVER (2) (PARM) do
    ...
  end WHATEVER (2);
  ...
end DO_IT;

task body DO_IT is
    ...
  for INDEX in 1..5 loop
    select
      accept WHATEVER (INDEX) (PARM);
      ...
      end WHATEVER;
    or
      null;
    end select;
  end loop;
end DO_IT;
```

The index expression on the **accept** statement is evaluated only when the **accept** statement is reached. A stand-alone indexed **accept** (one outside a **select**) has its index evaluated, and then it waits for a call to that particular family member. Entry families can be used to specify priorities among entry calls to closely related entries.

7.13 The abort Statement

One final statement that must be described with regard to Ada tasks is the **abort** statement. Although this statement is probably used only on rare occasions, implementers must always implement it. **Abort** statements specify the names of tasks whose execution is to be made

"abnormal." The rules that describe what abnormal means are complex. Essentially the task immediately becomes uncallable, and must become terminated no later than the next synchronization point.

7.14 The PRIORITY pragma

The Ada language uses the priority **pragma** to specify how a task is chosen for execution from the ready queue. That is, when one task relinquishes a processor, the ready task of highest priority is chosen to get it. This **pragma** is placed in the task specification. It statically assigns the specified priority level to the task in whose specification it appears. If the specification is a type statement, then all objects of that type have the same priority. Larger values of priority indicate higher levels of urgency. It is important to note that the priority of a task has no impact on the order in which **entry** calls are taken from a given entry queue. These are always taken in FIFO order. Furthermore, the priority of tasks need not effect the choice among open **accept** alternatives within a **select** statement. One final note on priorities of tasks is this: Ada semantics dictate preemptive scheduling of tasks. This means that if a task is executing and a task with a higher priority becomes available for execution (e.g., its delay has expired), the running task must be suspended and the task with higher priority must be given the processor.

7.15 Allocated Tasks

Tasks can be created by an object declaration or via use of an allocator for an access type variable whose object is a task type. The use of the **new** allocator with such a variable causes the dynamic allocation of a task object.